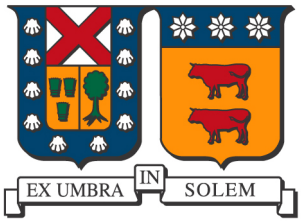
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



El enfoque de microservicios como estrategia para mejorar la calidad del software

Francisco Arévalo del Río
francisco.arevalod@alumnos.usm.cl

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERÍA DE EJECUCIÓN INFORMÁTICA

PROFESOR GUÍA MARCELLO VISCONTI
PROFESOR CORREFERENTE GASTÓN MÁRQUEZ
SEPTIEMBRE - 2016

Resumen

Al superar cierto tamaño, las aplicaciones se enfrentan a problemas de arquitectura que complican y demoran el desarrollo. El enfoque de microservicios es un patrón de diseño de software que aprovecha los protocolos de redes para comunicar pequeñas aplicaciones que cumplen un objetivo específico. Ésta arquitectura pretende resolver algunas de las falencias de las Arquitecturas Orientadas a Servicios (SOA), y ofrece una interfaz simple basada en Identificadores de recursos uniformes (URIs), y la metodología REST. Las técnicas propuestas se aplicaron a una aplicación monolítica, lo que permitió el uso de tecnologías más modernas y específicas. Se analizaron los pro y contra de este enfoque, y los beneficios en calidad, velocidad de desarrollo y desacoplamiento se comparan con la versión anterior. Se encontró que este enfoque permite realizar cambios de manera controlada, disminuyendo la cantidad de fallas y aumentando la velocidad de desarrollo.

Temas Relacionados: Arquitectura de Software, Calidad de Software, Patrones Arquitecturales, REST, Microservicios.

Abstract

When software projects go over a certain size, they face architectural challenges that slow down and hinder progress. The Microservices approach is a software design pattern that leverages the network stack and protocols to communicate small applications which fulfill a single business objective each. This architectural style addresses several criticisms that applied to Service Oriented Architecture, and offers a simple interface based on Uniform Resource Identifiers (URIs) and the Representational State Transfer design methodology. These techniques were applied to a monolithic application to enable the use of more modern and specific technology and tooling. Pros and cons to this approach are presented, and improvements to quality, development speed and uncoupling are compared to the former monolithic version. The result was a more controlled change process, a lower failure rate and better development velocity.

Keywords: Software Architecture, Software Quality, Architectural Patterns, REST, Microservices.

Índice

1. Introducción

Muchos proyectos de desarrollo de software se topan con problemas estructurales al ir creciendo y sobrepasar cierto tamaño crítico. La acumulación de código desechable produce una "gran pelota de barro"[3]. El crecimiento descontrolado genera acoplamiento entre las partes, dependencias abultadas, y variables de estado que se deben manejar, y que empiezan a acumularse a medida que el proyecto crece.

Una forma de controlar este aumento en la complejidad es utilizar patrones de diseño de software que permitan estructurar la aplicación, de manera de encapsular el código complejo dentro de sus abstracciones, y aumentar la seguridad con que se realizan cambios en el sistema.

Incluso así, cambios en los requerimientos, falta de conocimientos, o desinterés de los desarrolladores, terminan erosionando los diseños y arquitecturas. Si esto no se corrige, o al menos mitiga, se vuelve cada vez más dificil realizar cambios, hasta el punto en que podría ser necesario abandonar el proyecto y rediseñarlo desde cero.

Por lo tanto, los patrones que se apliquen deben ser tolerantes a los posibles cambios en la arquitectura, y ayudar a mantener el orden a medida que el proyecto crece. Son especialmente atractivos los patrones que permiten atacar un problema por partes, y separar responsabilidades entre los componentes, ya que evitan que se genere un componente “demasiado” complejo, desordenado, o con muchas reponsabilidades.

A medida que la complejidad aumenta, se hacen necesarios conceptos para describir lo que se está modelando, y herramientas que permitan operar sobre estas abstracciones. Dividir el problema en partes independientes permite seleccionar la herramienta precisa para cada situación, sin verse limitado por las decisiones tomadas en las otras partes del proyecto.

1.1. ¿Cómo estructurar un proyecto de software para que sea modular, testeable y escalable?

Aún cuando suenan razonables, alcanzar estos objetivos es un desafío; estructurar un proyecto siempre resulta mucho más complejo cuando se trata de un problema con las aristas de la vida real. Puede ser necesario realizar cambios en los modelos, agregar otros nuevos, e incluso volver a implementar partes de la solución que resultaron insuficientes. Otras veces no se cuenta con documentación ni diseños claros, o quizás ya se traen muchos problemas “históricos” a cuestas.

La metodología de microservicios se puede aplicar de forma sistemática para lidiar con estos problemas, identificando buenos puntos de división para tratarlos de forma independiente. Permite facilitar las pruebas que validan la funcionalidad, y simplificar el manejo de la carga en los distintos componentes, ya que no siempre es necesario optimizar la aplicación completa. Esto permite acercarse a los objetivos, modularidad, testeabilidad y escalabilidad.

Al separar las responsabilidades en servicios independientes, surgen interfaces claras para la interacción con otras partes de la aplicación, y es posible operar los componentes por separado, ya que son autosuficientes. Al tener funciones con objetivos explícitos y acotados, se simplifica el testing, ya que cada clase no depende de objetos y estados externos. Esto facilita el desarrollo de sistemas automáticos para garantizar la calidad del software.

1.2. ¿Cómo aprovechar los avances en instrumentación y tecnología, sin afectar al resto del proyecto?

Dividir el problema en microservicios permite introducir complejidad de forma controlada, ya que se puede partir con un sistema simple, respetando las interfaces entre los componentes, realizando cambios de forma gradual, y paulatinamente ir agregando detalles sin perturbar el resto del sistema. También ayuda a controlar esta complejidad y manejarla con las herramientas más apropiadas.

A medida que se desarrollan nuevas técnicas y adelantos, es favorable poder realizar cambios en las dependencias de un proyecto, aprovechar actualizaciones, y probar otras alternativas de implementación, sin tener que cambiar al mismo tiempo todo el código que podría estar relacionado. Esta es una ventaja de mantener separaciones bien definidas entre las partes de un proyecto de software, ya que al aislar los detalles de implementación internos de otras clases, es posible cambiarlos sin afectar a los demás componentes.

Las herramientas precisas para cada situación, como bases de datos y colas de mensajería especializadas, se pueden utilizar sin interferir con el resto de los sistemas, ya que cada microservicio tiene una implementación independiente.

1.3. ¿Cómo identificar factores que afectan la calidad del software, y evitar que afecten el proyecto?

Dependiendo de su motivación, su habilidad como desarrolladores, y la carga de trabajo a la que están sometidos, los miembros de un equipo pueden entregar resultados de calidad muy variable.

La metodología propuesta facilita el control de estos factores: simplifica el testing, aumenta su cobertura, y disminuye el tiempo que toma realizarlo. También permite aumentar la velocidad de desarrollo, reducir el tamaño del problema que cada componente resuelve, y facilitar la colaboración entre los miembros del equipo.

Hay muchas otras formas en que un proyecto se deteriora al pasar el tiempo, por ejemplo si las decisiones tomadas al principio limitan las opciones disponibles después, o al ser afectado por “dormant rot”, los cambios pasivos y del ambiente que afectan el software que no esta corriendo, o “active rot”, cuando las modificaciones al código comprometen la integridad de los datos, la documentación queda obsoleta, y al evolucionar se aleja del diseño original. Todas deben ser consideradas y mitigadas para evitar la pérdida de calidad.

2. Marco conceptual y teórico

2.1. ¿Qué es una metodología?

Un conjunto de Técnicas de representación y descripción, reglas de interpretación para esas técnicas (formas de aplicar), reglas de interconección (para combinar las técnicas y resolver casos más complejos), y heurísticas pasa su uso en general[8]. Esta definición crea una "caja de herramientas" que se aplica a los problemas a enfrentar usando esta metodología.

Practicamente todos los proyectos que no adoptan posturas claras frente a las técnicas a utilizar terminan implementando las mismas soluciones una y otra vez, hundidos en código redundante, de distintos niveles de calidad, y muchas veces parchando errores repetidos en cientos de lugares. En este caso, la metodología se refiere a la práctica de separar en servicios independientes, entregando la libertad de seleccionar las técnicas apropiadas a cada sub problema.

2.2. Refactoring

Martin Fowler plantea que una de las tareas principales al desarrollar software es mejorar el diseño del codigo existente, lo que bautiza como "Refactoring"[4]. El desafío es siempre encontrar el balance entre este trabajo de orden y mantención, y el desarrollo de nuevas funcionalidades. Lamentablemente, las presiones comerciales, ememergencias, y entropía hacen de esta tarea un desafío cada vez mayor.

La técnica que describe Fowler se basa en el ciclo "Red-Green-Refactor", en otras palabras, partir definiendo un test (que no se cumple, por eso "Red"), luego desarrollar el código necesario para cumplir el test ("Green"), para finalmente mejorar el diseño e implementación de cada componente.

Su propuesta es que refactorizar el código permite trabajar con abstracciones más claras y código más limpio, lo que se traduce en entrgas más rápidas y con menor cantidad de fallas.

Para poder aplicar estos principios, es necesario contar con cierta organización en el código, que permita separarlo en componentes individuales, con responsabilidades definidas y comprobables. Esto encaja con uno de los principios más importantes de la metodología de microservicios, y al seguirla se hace más facil aprovechar las ventajas de la técnica propuesta por Fowler.

2.3. Fielding y REST

Roy Fielding propone en su tesis doctoral una arquitectura para la comunicación de aplicaciones basada en los conocidos verbos HTTP, rutas que describen completamente el estado de la aplicación, y un servidor que no tiene concepto de memoria o estado; es decir, toda la información contenida en el request.[2]

Esto permitió el desarrollo de interfaces de comunicación estandard, generales, y universales. Si se diseñan servicios que sigan este patrón de diseño, es más facil operar entre sistemas, ya que las interfaces estan claramente definidas, hay formas estandarizadas de acceder a los recursos, y cada componente se hace cargo de sus responsabilidades de manera independiente.

Fielding aporta una pieza clave de la metodología: una forma para diseñar interfaces claras entre componentes en la web, sin importar si están implementados en distintos lenguajes, plataformas o servidores.

2.4. SOA

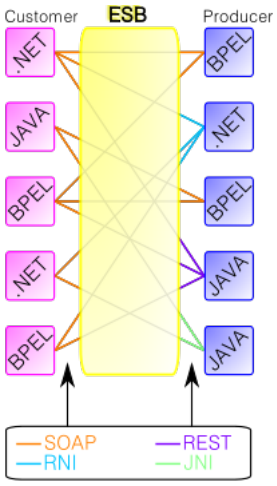


Figura 2.1: Bus de servicios que conecta procesos con cuatro protocolos distintos.

A principios de la década del 2000, alcanza popularidad un modelo llamado “Service-Oriented Architecture”, que tiene mucho en común con los microservicios. Lamentablemente, en vez de basarse en HTTP y el concepto de los URI (Uniform Resource Identifiers), no propone de un protocolo estandard de comunicación, sino que depende de un “Enterprise Service Bus”, que es el encargado de traducir y conectar los servicios con los consumidores.

Este componente resulta central para la implementación de una arquitectura distribuida, pero no se desarrolló un estandard que definiera las responsabilidades ni características que debía ofrecer.

Otra decisión de diseño que caracteriza esta metodología es su enfoque de orquestación y coordinación, a diferencia de los microservicios, donde el objetivo es la independencia. A medida que la tecnología de virtualización fue progresando, el costo de tener ambientes completamente separados (servidores, máquinas virtuales, contenedores) se redujo progresivamente, por lo que ya no hay necesidad de centralizar y compartir los recursos y dependencias.

La falta de estandarización en estos protocolos causó que surgieran varias alternativas contradictorias, por lo que muchos proyectos desarrollados según la metodología SOA invierten un esfuerzo considerable en traducir y comunicar sistemas y protocolos incompatibles. En el caso de REST, se aprovechó el estandard HTTP y las URI para evitar este problema, y todos los puntos de comunicación resultan compatibles entre si.

2.5. Virtualización, Contenedores y orquestación

Otro avance que cambió el panorama de la ingeniería de software fue el desarrollo de tecnologías de virtualización y contenedores, que permiten dividir un recurso computacional físico en uno o más ambientes independientes.

La encapsulación permite que la aplicación sea independiente del hardware en el que corre, controlar cuidadosamente su ambiente y configuración, y facilitar el despliegue y control de más instancias en pa-

ralelo. En vez de aprovisionar un servidor completo, basta con que se ejecute el contenedor o máquina virtual, y ya se cuenta con una nueva instancia independiente de la aplicación. Esto facilita el concepto de “Infraestrcutura como código”, y permite controlar, versionar y colaborar con el trabajo del equipo de operaciones.

En el caso de los contenedores, sólo se virtualiza el ambiente de ejecución mínimo para que la aplicación sea independiente, pero se comparten aspectos del sistema operativo, para permitir una ejecución más rápida y un contenedor más liviano.

De esta forma el problema de organizar y comunicar los procesos mediante un bus se transporta al nivel de comunicar contenedores o máquinas, lo que se facilita con el uso de una herramienta de orquestación. Ésta será responsable de descubrir, comunicar y asignar trabajo a los contenedores, pero aprovechando las tecnologías ya existentes de más alto nivel, sin tener que re implementar esta solución para cada aplicación y componente.

3. Exposición y análisis de un caso real y propuesta de solución

En esta sección se analiza el resultado de aplicar el patrón de microservicios a un producto de software real, diseñado para realizar tareas de facturación electrónica. Se exponen los problemas que enfrentaba el proyecto en un comienzo, seguidos por la aplicación de la propuesta, los cambios realizados, y los resultados obtenidos.

El programa maneja la información de usuarios, empresas, clientes, proveedores, prospectos, documentos tributarios, finanzas, pagos, cobros y contabilidad. Internamente, un conjunto de tablas en la base de datos, archivos que generan vistas en php, y un servidor web para ejecutarlos.

3.1. Presentación de los problemas que enfrenta un proyecto fuera de control

El diseño original de la aplicación fue evolucionando hasta incluir todas las tablas y conceptos mencionados, pero muchas veces no se tomó en cuenta la correcta integración de las nuevas funcionalidades con los sistemas ya existentes.

Ya que las responsabilidades de cada componente no estaban claramente definidas, se tiende a reimplementar operaciones ya desarrolladas en otra parte del sistema. En todos los módulos hay que obtener la información de los clientes, calcular sus balances, estados, etc. Esto no se desarrolló de forma coordinada, por lo que cada módulo implementa sus consultas de carga de clientes a su manera, muchas veces no considerando los cambios que han ocurrido en el modelo, ni las validaciones que hay que realizar. Un caso real era actualizar una consulta SQL, repetida en cientos de lugares con pequeñas variaciones, lo que resultaba casi imposible de automatizar.

Estructurado como proyecto monolítico, el código era muy dificil de reutilizar. El agregar una pequeña funcionalidad adicional podía tomar mucho tiempo, ya que muchos de los requerimientos previos eran ser desarrolladas para cada nuevo caso.

Esto lleva a cometer errores, a reducir la velocidad de desarrollo, y a frustración de los usuarios, al repetirse los problemas ya corregidos muchas veces dentro del sistema debido a la repetición de código.

Como inicialmente no se consideró una separación entre las capas de datos y presentación, resultaba muy dificil plantearse el desarrollo de una segunda interfaz (por ejemplo, una aplicación movil); habría que reimplementar toda la lógica de acceso a datos para aprovecharla en una nueva presentación. Si existiera un "servicio de datos", este tipo de problema se podría haber evitado.

3.2. Cuando los modelos no estan claramente definidos

La primera abstracción que se debe aplicar es el modelar el dominio del problema, definiendo las entidades, sus atributos y relaciones. Esto ya es un gran paso hacia una arquitectura más ordenada, ya que caracteriza los elementos base con que se organizará el resto de la solución.

Al explorar los modelos y sus relaciones, se debe tener especial cuidado al definir los métodos de acceso que se utilizarán, con el fin de que sean uniformes y reutilizables, y que cumplan todos los requerimientos del resto de las partes de la aplicación que interactúan con el objeto.

Si cada modelo define claramente como acceder, validar y modificar sus atributos, es posible reutilizar este concepto cuando sea necesario en otras partes de la aplicación. Cada entidad tiene responsabilidad de mantener sus valores, y realizar cualquier validación o manipulación propia a sus atributos. De esta forma se encapsula la complejidad, se evita el código duplicado (y los errores que trae), y se facilita el reutilizar los componentes.

3.3. Sistemas acoplados y con poco control

El siguiente síntoma se manifiesta comunmente cuando se ha ignorado el paso anterior. Al repetir funcionalidades y no controlar los accesos ni validaciones, se vuelve muy dificil mejorar o corregir algo que se encuentra repetido y repartido por toda la base de código.

Cuando una funcionalidad se encuentra mal encapsulada, y depende o modifica el resultado de otra, los cambios tienden a causar fallas en cadena. Los errores ya corregidos vuelven a aparecer, al ejecutarse un camino de código que no fue parchado, y las malas prácticas se contagian, al haber ejemplos negativos sembrados por el código.

Si no se controlan las dependencias y acoplamiento del sistema, es imposible realizar un cambio en el stack de tecnologías, ya que todo depende directamente del acceso directo a los recursos más básicos.

El acoplamiento genera otro gran problema cuando se diseña una estrategia de testing. Cuando hay dependencias complejas, generar un caso de testing puede ser casi imposible, ya que requiere organizar una gran cantidad de condiciones de entorno.

En el ejemplo de la facturación electrónica, la generación de un documento tributario requiere conseguir bloqueos sobre varias tablas, y coordinar la carga de entidades y variables para entregar al generador de documentos. Esta era una sola función que abarcaba varias páginas de carga y validaciones de datos. Tambien resultaba imposible de testear ya que tenía un sólo punto de entrada, que requería todos los parámetros, y un resultado de salida, que era un documento tributario generado.

Si este proceso se hubiera diseñado con el objetivo de mantener el acoplamiento al mínimo, de seguro habría tomado la forma de varios procesos interconectados que obtienen y validan la información a entregar a la siguiente etapa de la cadena. De esta forma, sería posible probar cada etapa intermedia, al tratarse de un número menor de variables de entrada, y un resultado de salida parcial, observable y más facil de analizar.

3.4. Balanceo de carga

Cuando toda la aplicación corre en un servidor monolítico, el balanceo de carga puede ser complicado, ya que “clonar” este servidor no será trivial. Si se genera inestabilidad, habría que comunicar el estado completo de la ejecución a un servidor de reemplazo, y redireccionar a todos los clientes al nuevo servidor. En cambio, si cada servicio se ejecuta por separado, se vuelve mucho más simple utilizar un balancedor de carga y contar con múltiples servicios en modalidad round robin.

Al estructurar la aplicación como servicios independientes, todo está pensado para poder comunicar todo el estado necesario por medio de llamadas, lo que permite reemplazar un servidor inestable sin impactar la disponibilidad del servicio ni la experiencia de otros usuarios en la plataforma. Basta con redirigir estos

mensajes a un servidor nuevo.

Además, la división natural de las bases de datos según cada servicio actúa como un balanceo de carga natural, repartiendo la carga de trabajo entre varios sistemas, que pueden escalar de forma independiente.

3.5. Alta disponibilidad y redundancia

Al no dividir los servicios correctamente, la aplicación completa debe ser replicada simultaneamente. Existe miedo al deployment, miedo al realizar cambios. Se debe actualizar el sistema completo de manera simulatanea, lo que afecta la disponibilidad del servicio. El sistema puede ser lento para partir, tener ciertas necesidades de cache, y el impacto de los cambios en el rendimiento siempre es incierto. Una aplicación monolítica cae de golpe.

Al considerar las garantías de servicio que establece un contrato comercial, también hay una gran ventaja al usar microservicios, ya que la capacidad redundante requerida será mucho menor que en el caso de necesitar un servidor más poderoso que corra todos los servicios, y el aprovisionar máquinas extra será más barato, simple y rápido si estas realizarán una menor carga de trabajo. De esta forma es posible ofrecer la garantía de alta disponibilidad, a un costo cada vez más razonable.

3.6. Dificultad para escalar

Al enfrentarse con la necesidad de escalar, existen dos alternativas para hacerlo: horizontal y vertical. Ambas se facilitan al aplicar un enfoque de microservicios. En el caso del escalamiento vertical, se trata de aumentar la capacidad de los recursos disponibles, por ejemplo, duplicar la memoria RAM, procesadores y espacio en disco del servidor.

Cuando la carga de la aplicación se concentra en una parte acotada del código, se desaprovechan muchos recursos si se sigue una estrategia de escalamiento vertical, ya que probablemente se irán descubriendo nuevos cuellos de botella a medida que se agregan recursos, y estos pueden ser caros de resolver simultaneamente en un solo servidor. Una parte de la aplicación puede depender del CPU, y otra del RAM, por lo que aprovisionar un servidor con gran capacidad de ámbos puede ser muy costoso.

Cuando cada servicio se aisla, es simple otorgar recursos a cada uno por separado, y asi aprovechar de forma más cuidadosa el presupuesto disponible. En el caso del escalamiento horizontal, se intenta agregar más servidores en paralelo y distribuir la carga entre máquinas similares.

Al dividir en servicios, cada máquina puede ser más pequeña, independiente, y se puede aprovisionar capacidad extra exclusivamente para los servicios más demandados. Esto ayuda a disminuir el costo y mejorar el rendimiento de la aplicación. Incluso es posible aprovechar distintas plataformas, hardware y arquitecturas para resolver cada parte del problema.

3.7. Aplicación de la propuesta

La aplicación se reestructuró gradualmente para adaptarla a una arquitectura de microservicios, separando las responsabilidades de cada módulo y haciéndolos más independientes, como se puede ver en la

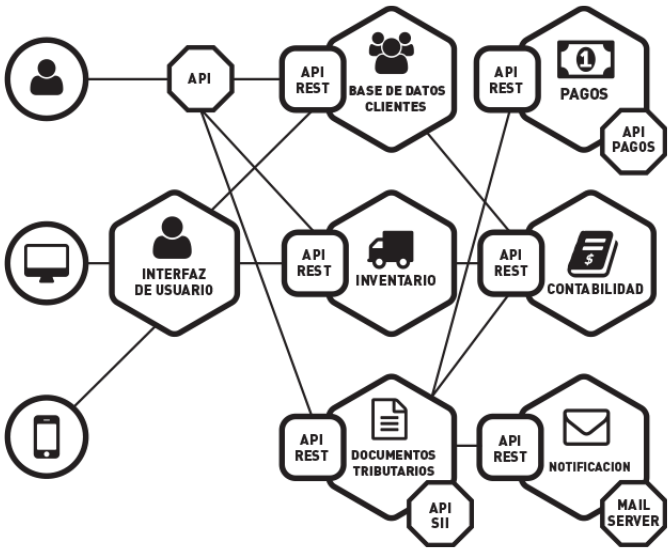


Figura 3.1: Arquitectura de microservicios separada por responsabilidades. Cada componente expone su API y delega subtareas al servicio responsable.

figura 3.1.

La primera ventaja al aplicar la metodología fue la aparición de modelos de dominio claros, con un punto único donde se definieron las operaciones de carga de datos, validaciones, consultas y filtros. Esto permitió reutilizar estas definiciones, y eliminar mucho código repetido, que era fuente de errores, fallas repetitivas y pequeñas discrepancias en los resultados.

La separación de los modelos de acceso a los datos permitió luego enfocarse en mejorar la capa de presentación, y mejorar la forma en que los datos se mostraban al cliente. Al contar con una representación clara y consistente para cada recurso, fue posible diseñar nuevas vistas y reportes, sin tener el problema de la variabilidad de los resultados que tenía cada consulta específica. Como cada vista fue diseñada basándose en recursos bien definidos, fue posible generar tests que garantizaran los supuestos en que depende la presentación de los datos.

Uno de los objetivos principales al planificar la migración fue mejorar la estabilidad en la generación de documentos, ya que el proceso era intensivo en carga, y se comunicaba con una API externa. Esto se consiguió aislando el componente que genera los documentos en un servidor y base de datos distintos, para evitar interferencia con el resto del sistema, y permitir que creciera de forma independiente en caso de ser necesario.

Al reducir el tamaño de cada componente, se hicieron más faciles de perfeccionar, permitiendo a los equipos de desarrollo trabajar de manera más independiente, y con la seguridad de no afectar otras partes del proyecto al realizar cambios. Esto se tradujo en mejoras de velocidad de desarrollo, disminución de fallas, y mayor libertad para tomar decisiones.

Por otra parte, algunos aspectos se volvieron más delicados al tratarse de una aplicación distribuida.

Para analizar los logs de la aplicación, hubo que diseñar un sistema para recolectarlos y unificarlos en una base de datos de logs. Afortunadamente, existen muy buenas herramientas e instrumentación para lograr este objetivo, y no fue necesario realizar un desarrollo muy profundo, ya que es un problema bien identificado por la comunidad de microservicios.[7]

El manejo de los servidores también sufrió cambios, pero estos fueron mayoritariamente positivos. A pesar de que las máquinas aumentaron en número, el proceso de aprovisionamiento, despliegue del código fuente y monitoreo se simplificó, y la carga en cada una resultó más manejable y pareja.

4. Resultados, Lecciones aprendidas y Mejoras al proceso

- La metodología de microservicios permite enfrentar estos problemas, entrega herramientas para controlar la creciente complejidad y dividir las responsabilidades de forma que puedan ser controladas. Al establecer límites e interfaces claras entre componentes, se pueden ir especializando y desacoplando.

- La funcionalidad total no cambia (refactoring), pero el problema queda dividido en partes abordables, más fáciles de entender, explicar, simplificar.

- Especialización de los equipos. No es necesario entender el sistema completo para aportar. Libertad de selección de tecnologías y herramientas, con la mentalidad "right tool for the job". Escoger las herramientas correctas es clave para permitir el avance sobre el problema principal.

- Independencia al deployar, escalar y testear. El server no tiene estado (REST). Es posible administrarlos como un recurso facil de aprovisionar y reemplazar ("ganado, no mascotas"), e incluso aprovechar máquinas virtuales y contenedores "desechables" para correr las aplicaciones.

- Es posible acostumbrarse a las restricciones tradicionales de SOA (mediación, ruteo, enriquecimiento de mensajes y traducción de protocolos) – pero quizás es mejor tener "Smart Endpoints and Dumb Pipes", y abstraer la lógica y responsabilidades de orquestación y despacho de mensajes, para no tener que implementarla en cada componente. Al depender de un "ESB", se genera un punto de falla central, donde se vuelve imposible operar el sistema sin un intermediario que traduzca e intermedie los componentes. En el caso de REST, cada módulo o servicio expone las rutas que ofrece, y se aprovechan protocolos comunes de comunicación. Así el problema se transforma en el descubrimiento y coordinación de máquinas o contenedores independientes, que cuentan con herramientas mucho más avanzadas para realizar esta coordinación.

- Algunas de las lecciones aprendidas se aplican a factores humanos que rodean el proceso de desarrollo: ¿quién debe hacerse cargo de cada servicio?, estará dispuesto el equipo a adaptarse a la forma de desarrollo (es un factor clave). Si la labor de identificar los patrones del problema no se realiza bien se puede terminar con una gran cantidad de pequeños servicios inútiles.

- A cambio de simplificar algunas tareas, otras se hacen más complicadas: : ¿Cómo seguir el estado de una petición compleja que abarca distintos servicios? ¿Cómo hacer calzar el patrón en una metodología de Integración Continua / Distribución Continua? ¿Cómo distribuir todos los servicios de manera coordinada? Hay que majear un ejército de servidores en vez de uno sólo, por lo que es indispensable contar con la instrumentación correcta para esto. Afortunadamente hay grandes avances teóricos y prácticos en esta área (Protocolos de coordinación, descubrimiento de servicios, consenso)

- Es fundamental disponer de un proveedor confiable de recursos computacionales. Existen excelentes proveedores de servidores privados virtuales, private clouds, como Amazon Web Services, Linode, y muchos otros. También hay buenas herramientas para controlarlos, como Kubernetes para la

orquestación, Ansible, Chef, o Puppet para el manejo de configuraciones, y Docker para ejecutar contenedores.

- Un buen sistema de logs y registro puede marcar la diferencia entre unas pocas horas y meses para resolver un problema. Pero con los microservicios, no resulta trivial coordinar y sincronizar los esfuerzos de logging. Este es un problema muy serio, que debe ser considerado desde un comienzo en el diseño. También existen buenas herramientas de logs distribuidos, que se encargan de recolectar, ordenar y filtrar desde varias fuentes remotas.

5. Conclusiones

En todos los proyectos de software, las decisiones de arquitectura le dan forma a las ventajas y limitaciones que tendrá su implementación. Cada funcionalidad ofrecida tiene su costo, en diseño, documentación y mantención.

Si no se realiza un esfuerzo conciente de diseño, buscando el orden y la organización lógica de los componentes, es imposible que el proyecto crezca más allá de un nivel básico. Si los factores de desorden identificados no son mitigados, causarán el deterioro del software tanto pasiva como activamente.

Asimismo, los buenos patrones de diseño ayudan a reutilizar, ordenar, mejorar y mantener el código. Esto fomenta las buenas prácticas, lo que a su vez permite concentrarse en el objetivo principal a resolver, y no distraerse en los detalles de la implementación.

Hoy en día la capacidad de procesamiento disponible y el bajo costo del hardware permiten separar los componentes de una aplicación en servicios independientes, y se dispone de un buen estandar y herramientas para comunicarlos y reutilizarlos.

Por estos motivos, el patrón de microservicios resulta ideal para aplicar hoy en día en todo tipo de proyectos, ya que permite testear, escalar y mejorar la aplicación de forma organizada e independiente, e incentiva el desarrollo de servicios con responsabilidades claras y acotadas.

6. Glosario

REST: Representational State Transfer. Estilo de desarrollo donde el servidor no guarda el estado de la aplicación.

SOA: Service Oriented Architecture, Arquitectura orientada a servicios.

URI: Uniform Resource Identifiers. Nombre Identificador de un recurso, generalmente su ubicación en la web.

HTTP: Hypertext Transfer Protocol. Protocolo usado en la web para transferencia de información.

API: Application Programming Interface. Definición de rutinas y protocolos para interactuar con una aplicación.

Referencias

[1] Allamaraju, S. (2010). RESTful Web services cookbook. Sebastopol, CA: O'Reilly Media.

[2] Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. Irvine, CA: University of California

[3] Foote, B., & Yoder, J. (1997). Big Ball of Mud. Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97). Monticello, Illinois

[4] Fowler, M., & Beck, K. (1999). Refactoring: Improving the design of existing code. Reading, MA: Addison-Wesley.

[5] Fowler, M., & Lewis, J. (2014). Microservices: Common characteristics of this architectural style. Disponible en http://martinfowler.com/articles/microservices.html

[6] Mauro, T. (2015). Adopting Microservices at Netflix: Lessons for Architectural Design. San Francisco, California.

[7] Newman, S. (2015). Building microservices. Sebastopol, CA: O'Reilly Media.

[8] Wieringa, R. (1998). A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. New York City, NY: University of Twente, ACM Computing Surveys, Vol. 30, No. 4, December 1998